PROCESSOR THAT ACCOMODATES MULTIPLE INSTRUCTION SETS AND MULTIPLE DECODE MODES

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority to U.S. Provisional Application Serial No. 60/400,391 titled "JSM Protection," filed July 31, 2002, incorporated herein by reference. This application also claims priority to EPO Application No. 03291917.7, filed July 30, 2003 and entitled "Processor That Accommodates Multiple Instruction Sets And Multiple Decode Modes," incorporated herein by reference. This application also may contain subject matter that may relate to the following commonly assigned co-pending applications incorporated herein by reference: "System And Method To Automatically Stack And Unstack Java Local Variables," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35422 (1962-05401); "Memory Management Of Local Variables," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35423 (1962-05402); "Memory Management Of Local Variables Upon A Change Of Context," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35424 (1962-05403); "A Processor With A Split Stack," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35425(1962-05404); "Using IMPDEP2 For System Commands Related To Java Accelerator Hardware," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35426 (1962-05405); "Test With Immediate And Skip Processor Instruction," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35427 (1962-05406); "Test And Skip Processor Instruction Having At Least One Register Operand," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35248 (1962-05407); "Synchronizing Stack Storage," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35429 (1962-05408); "Methods And

Apparatuses For Managing Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35430 (1962-05409); "Write Back Policy For Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35431 (1962-05410); "Methods And Apparatuses For Managing Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35432 (1962-05411); "Mixed Stack-Based RISC Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35433 (1962-05412); "System To Dispatch Several Instructions On Available Hardware Resources," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35444 (1962-05414); "Micro-Sequence Execution In A Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35445 (1962-05415); "Program Counter Adjustment Based On The Detection Of An Instruction Prefix," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35452 (1962-05416); "Reformat Logic To Translate Between A Virtual Address And A Compressed Physical Address," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35460 (1962-05417); "Synchronization Of Processor States," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35461 (1962-05418); "Conditional Garbage Based On Monitoring To Improve Real Time Performance," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35485 (1962-05419); "Inter-Processor Control," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35486 (1962-05420); "Cache Coherency In A Multi-Processor System," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35637 (1962-05421); "Concurrent Task Execution In A Multi-Processor, Single Operating System Environment," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35638 (1962-05422); and "A Multi-Processor Computing System Having A Java Stack Machine And A RISC-Based Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35710 (1962-05423).

BACKGROUND OF THE INVENTION

Technical Field of the Invention

[0002] The present invention relates generally to processors and more particularly to a processor capable of changing a decoder between modes according to multiple instruction set.

Background Information

[0003] Many types of electronic devices are battery operated and thus preferably consume as little power as possible. An example is a cellular telephone. Further, it may be desirable to implement various types of multimedia functionality in an electronic device such as a cell phone. Examples of multimedia functionality may include, without limitation, games, audio decoders, digital cameras, etc. It is thus desirable to implement such functionality in an electronic device in a way that, all else being equal, is fast, consumes as little power as possible and requires as little memory as possible. Improvements in this area are desirable.

BRIEF SUMMARY OF THE PREFERRED EMBODIMENTS

[0004] As disclosed herein, a processor (e.g., a co-processor) includes decode logic capable of decoding instructions from a first instruction set in a first mode and capable of decoding instructions from a second instruction set in a second mode. The processor also includes pre-decode logic capable of pre-decoding instructions prior to the loading of the instructions into the decode logic. In accordance with at least some embodiments of the invention, the decode logic may operate in a first mode and decode instructions from a first instruction set. The pre-decode logic also may pre-decode instructions concurrently with the decode logic. In particular, the pre-decode logic determines if an instruction is intended to temporarily change the decode logic from

the first mode to the second mode. For example, the mode changing instruction may comprise a prefix to a following instruction indicating that the following instruction belongs to another instruction set, for example, the second instruction set. In response, the decode logic switches from the first mode to the second mode to decode the following instruction.

[0005] In other embodiments, the decode logic may permanently change from one mode of operation to another. For example, the decode logic may operate in a first mode while the pre-decode logic pre-decodes an instruction (e.g., a prefix) that indicates that a plurality of instructions from the second instruction set are to be decoded. The prefix may indicate that the following instruction belongs to the second instruction set. Another instruction may follow indicating the presence of a plurality of instructions from the second instruction set. As such, the decode logic "permanently" switches from the first mode to the second mode to decode the plurality of instructions. "Permanently" means that the decode logic decodes instructions in only one mode until commanded otherwise.

NOTATION AND NOMENCLATURE

[0006] Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, computer companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to...". Also, the term "couple" or "couples" is intended to mean either an indirect or direct connection. Thus, if a first device couples to a second device, that connection may be through a direct connection, or through an indirect connection via other devices and connections.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] For a more detailed description of the preferred embodiments of the present invention, reference will now be made to the accompanying drawings, wherein:

[0008] Figure 1 shows a diagram of a system in accordance with preferred embodiments of the invention and including a Java Stack Machine ("JSM") and a Main Processor Unit ("MPU");

[0009] Figure 2 shows a block diagram of the JSM of Figure 1 in accordance with preferred embodiments of the invention;

[0010] Figure 3 shows various registers used in the JSM of Figures 1 and 2;

[0011] Figure 4 illustrates decoding an instruction in a first mode in parallel with pre-decoding a plurality of instructions; and

[0012] Figures 5 illustrates decoding in the second mode.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0013] The following discussion is directed to various embodiments of the invention. Although one or more of these embodiments may be preferred, the embodiments disclosed should not be interpreted, or otherwise used, as limiting the scope of the disclosure, including the claims, unless otherwise specified. In addition, one skilled in the art will understand that the following description has broad application, and the discussion of any embodiment is meant only to be exemplary of that embodiment, and not intended to intimate that the scope of the disclosure, including the claims, is limited to that embodiment.

[0014] The subject matter disclosed herein is directed to a programmable electronic device such as a processor. The processor described herein may be particularly suited for executing Java™ Bytecodes, or comparable code. As is well known, Java is particularly suited for embedded applications and is a relatively "dense" language meaning that on average, each instruction may perform a large number of functions compared to many other programming languages. The dense nature of Java is of particular benefit for portable, battery-operated devices that preferably include as little memory as possible to save space and power. The reason, however, for executing Java code is not material to this disclosure or the claims that follow. The embodiment of the invention may be described in the context of Java but should not limit the execution of only Java instructions. The processor described herein may be used in a wide variety of electronic systems.

[0015] Referring now to Figure 1, a system 100 is shown in accordance with a preferred embodiment of the invention. As shown, the system includes at least two processors 102 and 104. Processor 102 is referred to for purposes of this disclosure as a Java Stack Machine ("JSM") and processor 104 may be referred to as a Main Processor Unit ("MPU"). System 100 may also include memory 106 coupled to both the JSM 102 and MPU 104 and thus accessible by both

processors. At least a portion of the memory 106 may be shared by both processors meaning that both processors may access the same shared memory locations. Further, if desired, a portion of the memory 106 may be designated as private to one processor or the other. System 100 also includes a Java Virtual Machine ("JVM") 108, compiler 110, and a display 114. The JSM 102 preferably includes an interface to one or more input/output ("I/O") devices such as a keypad to permit a user to control various aspects of the system 100. In addition, data streams may be received from the I/O space into the JSM 102 to be processed by the JSM 102. Other components (not specifically shown) may be included as well.

[0016] As is generally well known, Java code comprises a plurality of "Bytecodes" 112. Bytecodes 112 may be provided to the JVM 108, compiled by compiler 110 and provided to the JSM 102 and/or MPU 104 for execution therein. In accordance with a preferred embodiment of the invention, the JSM 102 may execute at least some, and generally most, of the Java Bytecodes. When appropriate, however, the JSM 102 may request the MPU 104 to execute one or more Java Bytecodes not executed or executable by the JSM 102. In addition to executing Java Bytecodes, the MPU 104 also may execute non-Java instructions. The MPU 104 also hosts an operating system ("O/S") (not specifically shown), which performs various functions including system memory management, the system task management that schedules the JVM 108 and most or all other native tasks running on the system, management of the display 114, receiving input from input devices, etc. Without limitation, Java code may be used to perform any one of a variety of applications including multimedia, games or web-based applications in the system 100, while non-Java code, which may comprise the O/S and other native applications, may still run on the system on the MPU 104.

[0017] The JVM 108 generally comprises a combination of software and hardware. The software may include the compiler 110 and the hardware may include the JSM 102. The JVM may include a class loader, Bytecode verifier, garbage collector, and a Bytecode interpreter loop to interpret the Bytecodes that are not executed on the JSM processor 102.

[0018] In accordance with preferred embodiments of the invention, the JSM 102 may execute at least two instruction sets. One instruction set may comprise standard Java Bytecodes. As is well known, Java is a stack-based programming language in which instructions generally target a stack. For example, an integer add ("IADD") Java instruction pops two integers off the top of the stack, adds them together, and pushes the sum back on the stack. In general, the JSM 102 comprises a stack-based architecture with various features that accelerate the execution of stack-based Java code.

[0019] Another instruction set executed by the JSM 102 may include instructions other than standard Java instructions. In accordance with at least some embodiments of the invention, such other instruction set may include register-based and memory-based operations to be performed. This other instruction set generally complements the Java instruction set and, accordingly, may be referred to as a complementary instruction set architecture ("C-ISA") such as those instructions disclosed in one or more of the previously listed co-pending applications included herein by reference. By complementary, it is meant that at least some Java Bytecodes may be replaced by micro-sequences using C-ISA instructions that permit address calculation to readily "walk through" the JVM data structures. A micro-sequence may comprise one or more C-ISA instructions. Further, such micro-sequences may also use Bytecode instructions. The execution of Java may be made more efficient and run faster by replacing some sequences of Bytecodes by preferably shorter and more efficient sequences of C-ISA instructions. The two sets of instructions

may be used in a complementary fashion to obtain satisfactory code density and efficiency. As such, the JSM 102 generally comprises a stack-based architecture for efficient and accelerated execution of Java Bytecodes combined with a register-based architecture for executing register and memory based C-ISA instructions. Both architectures preferably are tightly combined and integrated through the C-ISA.

[0020] Figure 2 shows an exemplary block diagram of the JSM 102. As shown, the JSM includes a core 120 coupled to data storage 122 and instruction storage 130. The core may include one or more components as shown. Such components preferably include a plurality of registers 140, including a program counter 160 and a status register 162, three address generation units ("AGUs") 142, 147, micro-translation lookaside buffers (micro-TLBs) 144, 156, a multi-entry micro-stack 146, an arithmetic logic unit ("ALU") 148, a multiplier 150, decode logic 152, instruction fetch logic 154, and pre-decode logic 158. In general, operands may be retrieved from data storage 122 or from the micro-stack 146, processed by the ALU 148, while instructions may be fetched from instruction storage 130 by fetch logic 154, pre-decoded by pre-decode logic 158, and decoded by decode logic 152. The address generation unit 142 may be used to calculate addresses based, at least in part on data contained in the registers 140. The AGUs 142 may calculate addresses for C-ISA instructions as will be described below. The AGUs 142 may support parallel data accesses for C-ISA instructions that perform array or other types of processing. AGU 147 couples to the micro-stack 146 and may manage overflow and underflow conditions in the micro-stack preferably in parallel. The micro-TLBs 144, 156 generally perform the function of a cache for the address translation and memory protection information bits that are preferably under the control of the operating system running on the MPU 104. The decode logic 152 and the pre-decode logic 158 may be adapted to execute both the standard Java instructions as well as the C-ISA instructions of

the system. The operation of the decode logic 152 and the pre-decode logic 158 will be described in more detail below.

[0021] Referring now to Figure 3, the registers 140 may include 16 registers designated as R0-R15. Registers R0-R3, R5, R8-R11 and R13-R14 may be used as general purposes ("GP") registers usable for any purpose by the programmer. Other registers, and some of the GP registers, may be used for specific functions. For example, registers R4 and R12 may be used to store two program counters. Register R4 preferably is used to store the program counter ("PC") and register R12 preferably is used to store a micro-program counter ("micro-PC"). In addition to use as a GP register, register R5 may be used to store the base address of a portion of memory in which Java local variables may be stored when used by the current Java method. The top of the micro-stack 146 is referenced in registers R6 and R7. The top of the micro-stack has a matching address in external memory pointed to by register R6. The values contained in the micro-stack are the latest updated values, while their corresponding values in external memory may or may not be up to date. Register R7 provides the data value stored at the top of the micro-stack. Registers R8 and R9 may also be used to hold the address index 0 ("AI0") and address index 1 ("AI1"). Register R14 may also be used to hold the indirect register index ("IRI"). Register R15 may be used for status and control of the JSM 102. As an example, the status/control bit may indicate if the JSM 102 is executing a "simple" instruction or a "complex" instruction through a micro-sequence. This bit controls in particular, which program counter is used R4 (PC) or R12 (micro-PC) to fetch the next instruction. A "simple" Bytecode instruction is generally one in which the JSM 102 may perform an immediate operation either in a single cycle (e.g., an "iadd" instruction) or in several cycles (e.g., "dup2_x2"). A "complex" Bytecode instruction is one in which several memory accesses may be required to be made within the JVM data structures for various verifications

(NULL pointer, array boundaries, etc.). In another example, the status/control bit may indicate if the JSM 102 is executing an instruction of a first instruction set, e.g., the standard Java instruction set, or executing an instruction of a second instruction set, e.g., the C-ISA instruction set. Because these data structures are generally JVM-dependent and thus may change from one JVM implementation to another, the software flexibility of the micro-sequence provides a mechanism for various JVM optimizations now known or later developed.

[0022] Referring again to Figure 2, as noted above, the JSM 102 may be adapted to process and execute instructions from at least two instruction sets. One instruction set includes stack-based operations and the second instruction set includes register-based and memory-based operations. The stack-based instruction set may include Java Bytecodes. Unless empty, Java Bytecodes pop data from and push data onto the micro-stack 146. The micro-stack 146 preferably comprises the top *n* entries of a larger stack that may be implemented in data storage 122. Although the value of *n* may be vary in different embodiments, in accordance with at least some embodiments, the size *n* of the micro-stack may be the top eight entries in the larger, memory-based stack. The micro-stack 146 preferably comprises a plurality of gates in the core 120 of the JSM 102. By implementing the micro-stack 146 in gates (e.g., registers) in the core 120 of the processor 102, access to the data contained in the micro-stack 146 is generally very fast, although any particular access speed is not a limitation on this disclosure.

[0023] The second, register-based, memory-based instruction set may comprise the C-ISA instruction set introduced above. The C-ISA instruction set preferably is complementary to the Java Bytecode instruction set in that the C-ISA instructions may be used to accelerate or otherwise enhance the execution of Java Bytecodes.

[0024] The ALU 148 adds, subtracts, and shifts data. The multiplier 150 may be used to multiply two values together in one or more cycles. The instruction fetch logic 154 generally fetches instructions from instruction storage 130. The instructions first may be pre-decoded by the pre-decode logic by 158 and then decoded by decode logic 152. Because the JSM 102 may be adapted to process instructions from at least two instruction sets, the decode logic 152 generally comprises at least two modes of operation, one mode for each instruction set. In particular, the decode logic unit 152 may include a "Java" mode in which Java instructions may be decoded and a "C-ISA" mode in which C-ISA instructions may be decoded. Of course, the modes depend on the particular instruction sets implemented which may be different from the instrument sets described herein. While decoding instructions from one instruction set in the corresponding mode, the decoding mode may be changed "temporarily" to permit a single instruction from the other opposite instruction set to be executed in the mode that corresponds to the other instruction set. The opposite instruction set instruction may be preceded by a byte such as the Java "Impdep1" byte to cause the temporary mode switches. In addition, the decoding mode may be changed "permanently" to permit a plurality of instructions from the opposite instruction set to be executed in the mode set that corresponds to the opposite instruction set. Such an instruction may be preceded by a byte such as a prefix. The prefix may be one or more implementation defined Java bytes, e.g., the Java "Impdep1" byte. The pre-decode logic 158 determines if an instruction is preceded by such bytes and consider this byte as an instruction prefix, as will be discussed below.

[0025] The data storage 122 generally comprises data cache ("D-cache") 124 and data random access memory ("D-RAMset") 126. Reference may be made to copending applications U.S. Serial Nos. 09/591,537 filed June 9, 2000 (atty docket TI-29884), 09/591,656 filed June 9, 2000 (atty docket TI-29960), and 09/932,794 filed August 17, 2001 (atty docket TI-31351), all of which are

incorporated herein by reference. The stack (excluding the micro-stack 146), arrays and non-critical data may be stored in the D-cache 124, while Java local variables, critical data and non-Java variables (e.g., C, C++) may be stored in D-RAM 126. The instruction storage 130 may comprise instruction RAM ("I-RAM") 132 and instruction cache ("I-cache") 134. The I-RAMset 132 may be used for "complex" micro-sequenced Bytecodes or other "micro-sequences or critical sequences of codes," as will be described below. The I-cache 134 may be used to store other types of Java Bytecode and mixed Java/CISA instructions.

[0026] In accordance with the preferred embodiments, the decode logic may be capable of decoding instructions in two modes. The first mode is for instructions of a first instruction set, e.g. standard Java instruction set. The decode logic 152 may also decode instructions from a second instruction set, e.g., C-ISA, in a second mode. In order to ensure the appropriate decode mode for the decoding of instructions, the pre-decode logic 158 may pre-decode the instructions prior to loading the instructions in the decode logic 152. In particular, the pre-decode logic may pre-decode for instructions defined by the implementation that indicates the mode the decode logic should operate in for at least a succeeding instruction.

[0027] Referring to Figure 4, a portion of a program may include a plurality of bytes 164 (labeled as bytes "A" through "G"). In Java, these bytes may be called "Bytecodes." Therefore, for simplicity, these bytes may be referred as Bytecodes in both instruction sets. One or more of those bytes form a Java instruction. The Java instruction set, therefore, may include variable length instructions. In C-ISA, the instruction length also may vary in length (e.g., 2 to 4 bytes) and, like the Java instructions, the length may only be determined by the decode logic 152. In accordance with the preferred embodiments, while the decode logic 152 is decoding Bytecode A of an instruction in first mode, where Bytecode A may be all or a portion of the instruction, the next five

Bytecodes preferably are provided to pre-decode logic 158 for pre-decoding. As illustrated, Bytecodes B through F are pre-decoded by the pre-decode logic 158 in parallel with the decoding of Bytecode A.

[0028] In a preferred embodiment, each of the first and second instruction sets includes an instruction that causes the decode logic 152 to switch temporarily from one mode to another for only one instruction. In particular, both the first and second instruction sets may include a Java reserved code such as the "Impdep1" Bytecode. Upon detecting the Java Impdep1 Bytecode, the decode logic 152 switches to the other decode mode to decode the one instruction belonging to the instruction set that follows the Java Impdep1 instruction. For example, if the decode logic is operating in the first mode and the Java Impdep1 instruction is detected, the decode logic switches from the first mode to the second mode to decode the one instruction from the second instruction set that follows the Java Impdep1 instruction. Similarly, if the decode logic is operating in the second mode and the Java Impdep1 instruction is detected, the decode logic switches from the second mode to the first mode to decode the one instruction from the first instruction set that follows the Java Impdep1 instruction.

[0029] Referring still to Figure 4, Bytecode A belongs to a first instruction set, e.g., a Java instruction. In order to determine the mode in which the decode logic 152 may operate for a next instruction, the pre-decode logic 158 preferably determines if any of Bytecodes B through F comprises a predetermined instruction indicating at least one succeeding instruction following Bytecode A belongs to the second instruction set, e.g., an instruction from the C-ISA set. The predetermined instruction may be a prefix to the succeeding instruction. A prefix, as used herein, is a Bytecode that indicates the type of instruction that follows the prefix. For example, a prefix (e.g., Impdep1) may be used as a prefix and may indicate that the instruction following the

Impdep1 instruction belongs to a different instruction set. As such, by first determining if an instruction belongs to a particular instruction set, the decode logic 152 may caused to adjust the decoding mode for decoding the instruction. Therefore, if the pre-decode logic 158 determines that Bytecode B is a Java Impdep1 instruction and the length of the currently decoded instruction is one byte corresponding the Bytecode A, the decode logic 152 may be caused to skip the decoding of Bytecode B and switch from the first mode to the second mode to decode Bytecode C, where Bytecode C may be all or a portion of an instruction. After the decoding of the succeeding instruction that may include Bytecode C, the decode logic 152 switches back to the first mode. As such, the Java Impdep1 instruction may temporarily switch the decode logic from the first mode to the second mode to decode one instruction from the second instruction set and may switch back to the first mode after the decoding process of that instruction.

[0030] Similarly, the decode logic 152 may decode Bytecode A, where Bytecode A belongs to the second instruction set. Concurrently with the decode logic decoding Bytecode A in the second mode, the pre-decode logic 158 may determine that Bytecode B, out of Bytecode B through F, is a Java Impdep1 instruction. The decode logic switches from the second mode to the first mode to decode Bytecode C and may switch back to the second mode after the complete decoding of the instruction that includes Bytecode C.

[0031] In a preferred embodiment, the decode logic 152 may be decoding a current instruction in the appropriate mode. A program counter 160 may be used to point to that current instruction. Concurrently, the pre-decode logic may determine a Java Impdep1 instruction indicating that a following instruction belongs to another instruction set. A program counter calculation logic (not specifically shown) may increment the program counter 162 to skip the Java Impdep1 instruction and point directly to the following instruction such that the decode logic 152 does not decode the

Java Impdep1 instruction. Thus, by pre-decoding predetermined instructions indicating an instruction set belonging to another set prior to loading such instruction into the decode logic 152, the decode logic 152 may decode an instruction from another instruction set in the appropriate mode without a time penalty. In the preferred embodiment, these predetermined instructions may be the same for both instruction sets as described above. In another embodiment each instruction set might have a specific instruction that is pre-decoded to switch to the other mode for the following instruction, as described below.

[0032] In a preferred embodiment, the second instruction set may include an instruction that may permanently switch the decode logic 152 from the first mode to the second mode. In particular, the second instruction set may include a "SetC" instruction that indicates that a set of instructions, e.g., a set of C-ISA instructions, may follow. In the preferred embodiments, the SetC instruction follows a Java Impdep1 instruction belonging to the first instruction set. Assuming, the decode logic 152 is operating in a first mode, upon detection of the Java Impdep1 instruction, the decode logic 152 temporarily switches from the first mode to the second mode for decoding the following instruction, which is the SetC instruction. Upon detection of the SetC instruction, the decode logic 152 is set to permanently decode in the second mode. For example, referring to Figure 4, the decode logic 152 is decoding Bytecode A, which may belong to a first instruction set, e.g., the Java instruction set, in a first mode. In parallel with the decode logic decoding Bytecode A, the pre-decode logic 158 may determine if at least one of the five subsequent Bytecodes, Bytecodes B through F, include a predetermined instruction. In one example, the pre-decode logic 158 determines that Bytecode B is a Java Impdep1 instruction and that Bytecode C is a SetC instruction. Referring now to Figure 5, upon determining that Bytecode C is a SetC instruction, the

decode mode remains in the second mode and begins decoding the instructions after the SetC instruction in the second mode.

[0033] The second instruction set may also include an instruction that permanently switches from the decode logic from the second mode to the first mode. In particular, the second instruction set may include a "SetJ" instruction indicating that a set of instructions, e.g., a set of Java instructions, may follow. Upon detection of the SetJ instruction, the decode logic 152 permanently switches from the second mode (which may be used for decoding C-ISA instructions) to the first mode (which may be used for decoding Java bytecodes). The decode logic remains in the first mode until a Java Impdep1 instruction is detected.

[0034] In a preferred embodiment, the decode logic may also remain in the second mode upon detecting a predetermined instruction by the pre-decode logic 158. In particular, the pre-decode logic 158 may detect a Java Impdep1 instruction belonging to the second instruction set, and may cause the decode logic 152 to switch temporarily to the first mode for the one instruction following the Java Impdep1 instruction. Upon completion of the decoding process in the first mode for the one instruction following the Java Impdep1 instruction, the decode logic 152 switches back to the second mode.

[0035] In some embodiments, the Java instruction set may include two Java implementation dependent and reserved codes, e.g., Impdep1 and Impdep2, for temporary and permanent decode mode switches. For example, Imdep1 may be used to switch to the second mode for one instruction and Impdep2 may be used to permanently switch to the second mode. As such, if the decode logic is decoding a Java instruction in the first mode, upon detecting either the Impdep1 or Impdep2 instruction, the decode logic switches to the second mode. Similarly, the C-ISA may include a pair of codes for temporary and permanent decode mode switches.

[0036] By pre-decoding for a predetermined instruction that indicates that at least a following instruction belongs to another instruction set, the decode logic 152 may switch to the appropriate mode to decode the following instruction. Furthermore, by pre-decoding for the predetermined instruction in parallel with the decoding process, the predetermined instruction need not be loaded into the decode logic 152 and thus, improve the cycle time of decoding instructions from two different instruction set.

[0037] While the preferred embodiments of the present invention have been shown and described, modifications thereof can be made by one skilled in the art without departing from the spirit and teachings of the invention. The embodiments described herein are exemplary only, and are not intended to be limiting. Many variations and modifications of the invention disclosed herein are possible and are within the scope of the invention. Accordingly, the scope of protection is not limited by the description set out above. Each and every claim is incorporated into the specification as an embodiment of the present invention.